DOCUMENT RESUME

ED 175 440 IR 007 607

AUTHOR Neches, Robert
TITLE Promoting Self-Discovery of Improved Strategies.
SPONS AGENCY Advanced Research Projects Agency (DOD), Washington, D.C.; National Inst. of Mental Health (DHEW), Bethesda, Md.
PUB DATE 24 Mar 78
GRANT ARPA-F44620-73-C0074; NIMH-MH07722
NOTE 46p.; Best copy available; Paper presented at the Annual Meeting of the American Educational Research Association (San Francisco, California, April 8-12, 1979)

EDRS PRICE MF01/PC02 Plus Postage.
DESCRIPTORS *Cognitive Processes; Complexity Level; *Discovery Learning; Learning Theories; *Mathematical Concepts; *Memory; *Problem Solving; *Task Analysis; *Thought Processes

ABSTRACT

This paper describes an approach to task analysis which seeks to identify potential sources of difficulty in the self-discovery of improved procedures by students who have been taught simpler procedures. The approach considers novices' procedures in terms of the changes needed to produce an expert procedure: the knowledge required to make those changes, and the processing demands of acquiring that knowledge, can then be determined. An analysis of strategy improvement in a relatively simple domain--single digit arithmetic--is provided as an example. (Author/RAO)

AERA Session #29.18

Promoting Self-Discovery of Improved Strategies

Robert Neches
Department of Psychology
Carnegie-Mellon University
24 March 1978

Abstract

Expert strategies are sometimes too complex to teach directly. Often, it may be more effective to teach a simpler procedure and let students discover the more efficient procedure for themselves. This paper describes an approach to task analysis which seeks to identify potential sources of difficulty in the self-discovery of improved procedures. The approach considers novices' procedures in terms of the changes needed to produce an expert procedure. The knowledge required to make those changes, and the processing demands of acquiring that knowledge, can then be determined. Some example analyses are presented.

Paper presented at the annual conference of the American Educational Research Association, San Francisco, California, April 8-12, 1979.

The research reported in this paper was supported in part by NIMH Grant #MH07722, and by ARPA Grant # F44620-73-C0074.

Table of Contents

1. Introduction

The efficient strategies used by experts are often quite complex, and therefore unsuitable for direct instruction. One instructional alternative in such cases is to initially present a simpler strategy, and then work to promote its transformation into the expert strategy. In order to do this, it is necessary to have a detailed understanding of the demands of acquiring the expert strategy.

Greeno & Simon (1974) and Simon (1975) both have pointed out that there are often several different, but "functionally equivalent", methods for performing a task. Even though these methods may all be sufficient, they make very different demands on an individual's processing and memory capacities. The demands of the particular method being used will determine not just a student's facility at a task, but also the ease of incorporating that skill as a sub-skill in more complex tasks.

Resnick (1976) has suggested that useful instructional task analyses must consider both the sophisticated methods which the student is eventually to master, and the best *sequence* of simpler strategies with which to lead the student to the expert strategy. She reports that, even in the simple task of performing single-digit subtractions, small children discover for themselves methods which would be very difficult to give them by direct instruction.

Neches (1979) has studied the process of incremental refinements to an initial procedure by which an expert strategy is developed. Neches & Hayes (1978), referring to this refinement process as "strategy transformation", note that a small set of categories are sufficient to classify most instances of strategy change. Each type of change requires attending to different aspects of the current procedure. Neches (1979) divides the change process into three phases: (a) *noticing* the possibility of making a change of a particular type; (b) *suggesting* exactly what form the change will take; and, (c) *establishing* the change permanently in the repertoire of procedures. He found that attempts to develop improvements could be abandoned in any of these three stages.

Information processing analyses of strategies for performing a task can be used to suggest how those strategies will change as a student gains experience with a task. This paper describes techniques for performing such analyses, with the goal of identifying potential sources of difficulty in discovering improvements to procedures.

I will start by developing a model of the strategy change process in section 2. This section will start with a set of assumptions about the basic structure of the human information processor, describe a set of mechanisms which constitute the heart of the model,

and then expand on the portions of the set relevent to task analysis.

The approach to task analysis I will be presenting involves carefully defining both the procedures to be developed and the initial novice strategies we expect to see employed. The goal procedure can then be considered in terms of the changes which must be made in order to produce it from the initial procedures. Making a change to a procedure requires noticing the appropriate properties of that procedure as one performs it. Different properties, if detected, suggest different types of changes; one of the model's goals is to specify these properties for each transformation type.

The task analyses focus on two major potential sources of difficulty: (a) ease of discovering essential changes; and (b) competition from alternative strategy changes. Determining the properties of the initial strategy provides some knowledge of the set of likely changes to it. Determining what must be noticed in order to discover one of those properties, in turn, makes it possible to analyze the *distribution of prerequisite information.* That is, we can look for the points in the procedure where critical items of information appear. By considering the number of operations intervening between those points, we can attempt to make predictions about the difficulty of making a particular change. The paper describes several factors affecting these predictions. For example, simple assumptions about capacity limitations in human short term memory play a role in predicting how difficulty of discovering a change will increase with the number of intervening operations.

The last section of the paper will attempt to illustrate these points by presenting analyses of strategy improvement in a relatively simple domain: single digit arithmetic.

2. A Model of the Processes Underlying Strategy Changes

The purpose of this section is to provide a theoretical foundation for the approach to task analysis developed in the sections following. An attempt is made here to offer a general model which will apply to a broad range of circumstances, and to specify that model as completely as possible. A complete information processing model should specify its assumptions about the information processing system, as well as its hypotheses about the processes which that system executes. Since I am addressing a mixed audience, the reader should note that the discussion which follows is likely to be of greater interest to psychologists than educational researchers. After reading the next few paragraphs, those with primarily applied interests can afford to skip to section 2.5.

2.1 An overview of the model

While executing a procedure, information about recent operations resides in working memory. There is a small set of basic methods for producing different kinds of changes to a procedure. Associated with each method are sets of patterns to be matched against working memory. When a match is found, a goal is established to apply the associated method using the information which matched the pattern. When such a goal is established, the system tries to find an appropriate modification to the current procedure. This may lead to additional changes being required for the procedure to continue to work correctly. However, I believe that the strategy change process seeks to minimize effort. If it is too difficult to generate a change, the attempt is abandoned.

In summary, the theory has five key assumptions. (A) It places a strong emphasis on short term memory capacity limitations. (B) It assumes that processing is primarily bottom-up rather than top-down. (C) It assumes that the system is highly heuristic -- transformation types and their associated patterns are not guaranteed to work, but are usually worth gambling on. (D) It assumes that the process is designed for minimal effort -- it gives up high performance at strategy improvment in return for reduced processing effort. (This is done by cutting-off processing which doesn't produce quick success.) (E) Finally, the theory assumes that strategy change is a background process -- people are always trying to do it, but they only use those resources left over from the task at hand.

2.2 Information in working memory

As a procedure is executed, information about it enters working memory. Since short term memory is effectively limited in its capacity, information can be retained only about relatively recent operations. "Information", as the term is used here, is defined very broadly. The minimal information which seems to be available consists of (a) the *actions* just

performed; (b) the data operated upon by those actions; and, (c) the resulting states produced by those actions. It seems that additional information can include a range of background knowledge (e.g., that addition is commutative and associative, that subtraction is not, and so on.)

There is also some evidence in protocols of subjects doing sequence generation that *anticipation information* is available to a limited degree. That is, in making changes to their procedures, subjects sometimes seem to be making use of knowledge about what they *will be doing*, in addition to their knowledge about what they *have just done*. This is not a surprising empirical discovery. It is, however, a common sense observation that should be accounted for by a theory of the representation and execution of procedures in the human mind. I would like to see a *production system* theory provide this account, since -- as I will argue below -- production systems provide a promising structure in general for models of learning. However, in most formulations of production systems (e.g., Newell & Simon, 1972; Newell, 1973), actions are selected only on the basis of the current contents of working memory; it is impossible to predict what the next action will be, because there is no way of knowing in advance how the current action will alter working memory.

One way to understand the availability of these different sources of information is to view short term memory in the light of spreading activation models (Anderson, 1976; Collins & Loftus, 1975; Levin, 1976). Rather than postulating the kind of separate structure proposed by Waugh & Norman (1965), and accepted by such theorists as Newell & Simon (1972), spreading activation models describe memory as a uniform semantic network. Nodes in the network may be more or less "active", with the activity level of a given node being some function of the activity levels of nodes linked to it. Though details vary, the models generally claim that short term memory corresponds to the set of currently active structures in the semantic network (i.e., those activated above a certain level).

2.3 Initiation of strategy changes

The first stage of strategy change is, in this model, based on pattern detection. More specifically, the model asserts that there is a set of *transformation types* -- methods for producing various kinds of changes in a procedure. Each transformation type has associated with it a set of patterns to be matched against the trace information in working memory. When a pattern is matched, a goal is established to apply the associated transformation type. These goals can be viewed as loosely analogous to the APPLY goals of GPS (Newell & Simon, 1972; Ernst & Newell, 1969) in that other operations may have to be performed first to enable performance of the goal operation.

2.3.1 Pattern matching control processes

This view leads to a concern with pattern matching processes. For the moment, a "pattern" will be defined to be simply a set of "conditions" or tests. A pattern is matched if all of its conditions are satisfied. In this model the patterns are compared to the trace information active in working memory. The tests specified in a pattern are conditions which each must be satisfied by an active node or group of nodes. The issues to be addressed in specifying a pattern matching model can roughly be divided into (a) assumptions about the *processing rules* governing pattern matching; and (b) hypotheses about the *contents of specific patterns* which are tested according to those rules. The latter are presented in section 2.5.

From a psychological standpoint, most assumptions about processing rules are *ad hoc*, since there is no empirical evidence to suggest preferences among the alternatives. Production systems, the most psychologically oriented pattern matching scheme, first began to receive major attention with Newell & Simon (1972), although the first really detailed application was Newell's (1973) model of Sternberg's (1969) memory scanning task. A number of production system models of performance at various tasks have been presented (Baylor & Gascon, 1974; Klahr & Wallace, 1970; Ohlsson, 1977; Young, 1973). However, aside from Newell (1979), only Klahr & Wallace (1976) and Anderson (1976) have tried to develop production systems as comprehensive information processing theories. Their work borrows heavily of concepts developed in Artificial Intelligence rather than psychology (Newell, 1973; Davis & King, 1975; Lenat & McDermott, 1977; Waterman and Hayes-Roth, 1978). In general, there are three important considerations: (a) how conditions in a pattern are compared against data; (b) how patterns are selected for testing; and, (c) how a choice is made among multiple pattern matches.

The major thing which needs to be said about the comparison of condition elements to data elements is that the process is highly prone to combinatorial explosion. The source is the need to compare every condition element to every data element if all match possibilities are to be considered. Clever programming can cut this explosion considerably -- for example, by testing conditions one at a time and eliminating all patterns containing conditions which fail (Forgy, 1978). However, while these methods reduce the force of the explosion, they cannot contain it. Any pattern-directed system which purports to be an information processing model accounting for real time behavior must explain how the combinatorics of condition-testing are handled. Thus, any theory of pattern matching must, of necessity, assume a great deal of parallel processing. In addition, it may be necessary to assume that heuristics are used to avoid testing all possible combinations of data elements and condition

elements.. This introduces an element of uncertainty; such a system would sometimes fail to notice that the data were available to match a pattern.

Similar considerations affect the selection of patterns for testing.

2.3.2 Consistency with learning theories

It is important to note that this structure necessarily predicts some equivalent to the laws of recency or contiguity propounded by both behaviorist and gestaltist learning theories (Guthrie, 1952, Hull, 1952; Koffka, 1943; Thorndike, 1913). All of these theories were concerned with the connection or association of two events, whether those events were referred to as stimuli and responses, or as gestalts. While differing on the roles of frequency and reinforcement, all agree on a principle of contiguity: stimuli must appear close together in time to be considered part of the same event.

How does a pattern matching model predict such a law? With no stronger premise than that short term memory is limited in capacity, it follows that the closeness in time of relevent information will affect the probability of a pattern being matched. Imagine a pattern with two conditions, which are capable of being satisfied by items I_1 and I_n respectively. (Remember that items represent pieces of information of the sort discussed in section 2.2.) The distance in time between I_1 and I_n is the number of operations intervening between their arrival in working memory.

At the extreme, the effect of distance is clear; if I_n is too far away, then I_1 will have dropped out of short term memory when I_n appears, so the pattern can not be matched. At closer intervals, so that I_1 and I_n are both active in working memory simultaneously, their distance may also still have an effect. In general, since most theories of short term memory view it as a queue, we can expect that *the time for which two items will both be in working memory will increase as the distance between them decreases.* If we make the simple assumptions that patterns are tested whenever working memory changes, and that there is some probability that a match will be ignored, then it follows that the longer the two items are both available, the greater the probability that the match will be found.

Even under much more complex assumptions, such as embodied in Anderson's (1976) ACT system, the same conclusion holds. In ACT, Anderson assumes that patterns are matched against the set of active nodes and links. Periodically, all members of this set are deactivated, with the exception of the ten nodes most recently assigned to a special "Active List". Patterns have a strength associated with them, which is a measure of how often they have been useful in the past. It is reasonable to say that the frequency with which a pattern is tested depends on its strength, although this simplifies the actual mechanisms of ACT

somewhat. This non-zero probability of ignoring a match is sufficient to predict a contiguity effect; the only difference Anderson's more complex assumptions make is that they predict that the size of the effect can vary.

2.4 Representation of strategy changes

Previous sections have presented an argument for a pattern-driven system as a perspicuous model of the processes underlying strategy changes. In this section, I will argue that the procedures being modified, and the changes being made to them, are also best represented within the same structure. Production systems are an especially suitable medium for representing acquired procedural knowledge.

The key idea underlying production systems is the pattern-action rule, or production. The basic cycle of a production system consists of (a) testing the pattern part of each production against the contents of a data memory; and then, (b) executing the action part of one or more productions whose pattern was matched.

Part of the case for the virtues of production systems is presented by Newell & Simon (1972, pages 804-806). Among the reasons they cite in support of production systems as theoretical constructs are: that each production is independent of the others, that the organization (e.g., a single, central working memory) clearly corresponds to well-defined psychological constructs, and that production systems struck "a nice balance" between what are these days called "event-driven" *vs.* "schema-driven" processing. The latter argument is part of a set of arguments that production systems have sufficient power to model key aspects of human behavior. The middle argument is part of a set of arguments that production systems are plausible simulations of the human information processing system. These are important, but the key argument is the first.

Productions are independent because they do not interact directly, but only through working memory. They do not call each other by name; the only way they can be linked is if the action of one places information in working memory which causes the pattern of another to be matched. Thus, the knowledge about its fellows embodied in a production is of an extremely limited nature. When a production performs an action, such as changing the contents of working memory, the *only* assumption made about other productions is that there is at least one which will respond appropriately. No assumptions are embodied in that production about which production will respond, nor about what action it will take.

This means that it should be relatively easy to develop programs which modify production systems, since changing a system requires only manufacturing a new production and adding it. The modification program does not need to understand the flow of control of

the program it modifies; it only needs to understand the small, local function of the production it is adding. This presents a much more constrained, manageable task for a learning system.

Until relatively recently, the argument for production systems as tools for modelling learning was entirely theoretical. In the last few years, however, a number of *self-modifying production systems* -- programs with the capacity to add production rules to themselves -- have been demonstrated, starting with Waterman's work on adaptive production systems (Waterman, 1975, 1976). Since his initial efforts, several self-modifying systems have been developed which learn problem solving strategies. Neves (1978), for example, has developed a LISP program which builds productions for simplifying algebraic equations, guided only by examples. His program gradually constructs a complete procedure, composed of productions induced from a series of different examples.

Yuichiro Anzai, formerly of Carnegie-Mellon University and now at Keio University, has demonstrated a program quite similar in spirit to the theory presented here. His system acquires increasingly sophisticated strategies for solving the Tower of Hanoi puzzle (Anzai, 1978). The system initially starts out with a set of productions for solving the puzzle by heuristic search, and a set of productions for adding new productions. As moves are tried in the heuristic search stage, some of the latter productions discover that certain classes of moves were unwise and build productions to avoid them in the future. These new productions give the system a new strategy for solving the puzzle in which moves are selected by eliminating all unacceptable alternatives, falling back upon heuristic search only if multiple alternatives remain. From there, the system is able to build productions which represent an induced set of goals and subgoals, which in turn enable it to build up small programs for certain sub-tasks of the problem. Anzai's work represents a very clear demonstration of the power of production systems for learning systems in which procedures are built by piece-wise development of component parts.

The design of Anzai's program incorporates a certain amount of knowledge about the puzzle it is solving. Thus, it is difficult to evaluate the generality of his learning mechanisms. The principles used in his "bad-move elimination" productions seem most clearly general. In as yet unpublished work, he has used similar principles in a program which develops strategies for the Piagetian seriation task. We will be seeing very similar principles appearing in the discussion of a strategy transformation type which I call "deletion of unnecessary parts" (section 2.5).

2.5 Pattern detectors for seven transformation types

So far, the specification of the model has largely been concerned with describing a processing environment. This description has been stated as a set of assumptions about the *structure* of the information processing system in which procedures are executed and modified, the *control* of the processes which produce those changes, and the *representation* of procedures and their modifications. This section will attempt to further specify the model by detailing a set of strategy change mechanisms within this processing environment. I will be considering seven strategy transformation types proposed by Neches & Hayes (1978), and observed among subjects practicing at a complex symbol manipulation task (Neches, 1979):

1. *Reduction to results*: converting a computational process to a memory retrieval process.
2. *Reduction to a rule*: replacing a procedure with an induced rule for generating its results.
3. *Replacement with another method*: substituting an equivalent procedure obtained by noting analogies.
4. *Unit building*: grouping operations into a set accessible as a single unit.
5. *Deletion of unnecessary parts*: eliminating redundant or extraneous operations.
6. *Saving partial results*: retaining intermediate results which would otherwise have to be recomputed later in a procedure.
7. *Re-ordering*: changing the sequence in which operations are performed.

For each transformation type, I will propose one or more patterns associated with it. Neither the list of transformation types nor the sets of patterns are intended to be exhaustive. In understanding these patterns, it is crucial to keep in mind a key point: these patterns represent *heuristics* for discovering possible strategy changes. They serve to focus attention on some aspect of the process being carried out. The patterns suggest a likelihood of that aspect being interesting in terms of making some change, but they do not guarantee that a change will be either correct or worthwhile. The issues of determining the correctness and value of a strategy change are of concern in specifying the processes leading to adoption of changes. A discussion of those processes is well beyond the scope of this paper. Here, I am only concerned with the question of what might lead to a particular change being considered.

2.5.1 Reduction to results

It is important to distinguish between automatic and controlled processes associated with the learning of answers. I will assume very little about the automatic processes, although I have some favoritism for two: *strengthening* of frequently traversed links between nodes (Anderson, 1976), and EPAM-like *discrimination learning* (Feigenbaum, 1963).

However automatic learning takes place, it is still the case that memory for results can be the outcome of either automatic or goal-driven processing. Neches (1979) reports that subjects under instructions to learn to perform a task efficiently will identify portions of their procedure as important to memorize, and then rehearse results of those portions. Thus, in addition to incidental learning, there are cases where a goal is set to attend to certain information.

One pattern which would suggest that such a goal is worthwhile is this:

> A procedure recurs frequently, and the range of different inputs to it seems small.

The power of a pattern like this is extended by the ability of strategy changes to interact. Section 2.5.4, for example, describes a unit building pattern which forms new procedures out of actions which commonly occur together. This pattern would suggest learning the results of that new procedure, which provides the effect of learning the results of the original sequence of actions.

2.5.2 Reduction to a rule

Often, when a procedure is being observed over some range of inputs, it may be too difficult to learn a list of connections between inputs and results. However, it may be the case that some rule can be discovered about the relationship between inputs and results. If so, then that rule can be used instead of the procedure itself.

Patterns in the sequence of actions being performed may not serve to identify such rules. The patterns, though, can give very strong clues as to when it may be fruitful to actively search for such a rule.

The most general pattern which can be stated is the *correlation rule*:

> A procedure is observed for several inputs and their corresponding results. Whenever the input has some property, the result which is generated has some specific corresponding property.

To illustrate a trivial application of this pattern, imagine a small child learning to play a dice game which has the rule, "A player's marker may be moved only if either of the dice shows 2, 4, or 6." Imagine further that the child has not yet learned to recognize the dot patterns displayed on each face of a die. To determine whether a move can be made, the child must count the dots to see what number is there. A pattern of correlation can then be observed: whenever the dots can be formed into pairs, the answer is "yes", and whenever there is a dot left over, the answer is "no".

The correlation pattern for inputs and outputs has an analogue for effort:

> A difference in effort expended is observed when the same procedure is operating on the same input at different times.

This pattern suggests trying to find some factor which correlates with the effort difference. This factor can then be built as part of a rule, which seeks to ensure its presence when the procedure is executed.

This pattern is useful because procedures do not exactly duplicate themselves each time they are executed. They operate on objects which vary in their properties, they make random choices, errors are sometimes made, actions (especially motor responses) are not rigidly defined, and so on. All of this variability need not affect the correctness of a procedure, but it may lead to systematic variability in its efficiency.

One last pattern handles a useful set of special cases:

> A procedure is observed for a series of inputs, and each input is related to the previous one by being a successor in some known sequence.

If inputs are related in some orderly fashion, then it may be useful to try to determine if the corresponding results are also related. If so, then a rule can be developed where a procedure's output is predicted on the basis of its previous output (rather than on the basis of its current input).

2.5.3 Replacement with another method

Closely related to some of the reduction to a rule patterns are some patterns which suggest that some procedure is a suitable substitute for the procedure currently being used.

The patterns described here correspond to the similarity-detection methods proposed by Neches & Hayes (1978) called "result matching" and "description matching".

> A case is observed involving two different procedures, each of which operates on the same input and produces the same result, but one of which involves less effort.

This pattern leads to the discovery that two different procedures are equivalent. It naturally suggests that they are substitutable. We can state a similar pattern which represents a useful special case.

> The same procedure is observed to produce the same result for two *different* inputs.

In this case, the pattern suggests that one particular input is substitutable for another. If there is a marked difference in the effort required to apply the procedure to the two inputs, then it becomes worthwhile to adopt a policy of substituting the easier one.

Both of the above patterns represent result-matching rules. It is also possible, in an information processing system which keeps a network of knowledge associated with its procedures, to have descriptions of the goals which a procedure is intended to acheive, or the changes which a procedure is intended to produce. Analogous patterns to the ones stated above can then be stated for these semantic conditions.

2.5.4 Unit building

As series of actions are executed, it often becomes useful to group them into sets accessible as a single action. Gerritsen, Gregg, & Simon (1975), for example, taught different procedures for a symbol manipulation task which consisted of a number of sub-problems. They found a pattern of increased latencies between groups of sub-problems, and fast latencies within groups, in all of their subjects -- in spite of the fact that none of the methods taught predicted such a pattern.

This grouping process is the fundamental process of a number of learning theorists. It

has appeared most recently in the developmental theory of Klahr & Wallace (1976) as "consistent sub-sequence detection", and in Lewis' (1978) model of practice and *einstellung* effects as the process of "composition".

Basically, the pattern which suggests grouping is relatively straightforward:

> A procedure, P_1, is frequently followed by another procedure, P_2, and the result of P_1 is used by P_2.

2.5.5 Deletion of unnecessary parts

There are many different conditions which render a set of actions unnecessary, and therefore eliminable. It may be, for example, that:

> A sequence of connected procedures, $P_1,...,P_n$, is observed for which the output of the last (P_n) is identical to he input of the first (P_1).

This pattern is equivalent to the loop-move detectors of Anzai's (1978) learning program, which is discussed in section 2.4. It notes that the effects of operations early in a sequence have been undone by the later operations. Since the effect of the sequence is to take things back to where they started, the entire sequence serves no purpose.

Producing a change in the situation is necessary for a set of operations to be useful, but it is not sufficient. The result must contribute towards solution of the task at hand. We can describe one case where this requirement is violated by stating the following pattern:

> Some action contained in a procedure is observed to have *different* results at times when the procedure itself is observed to have the *same* results.

This pattern suggests that the action performed *has no influence* on the final outcome of the procedure in which it is carried out. Extraneous actions are, of course, unnecessary.

Rather than having no influence, an action may be unnecessary because its effects are predictable or redundant. The next two patterns deal with those cases:

> A decision is made about what to do next, on the basis of some action's result; that action, however, is observed to have a constant result.

If a decision is always the same, it need not be made. This pattern describes the case where a procedure is overly general. That is, the procedure spends time considering hypothetical possibilities which rarely occur in practice. Bhaskar & Simon (1977) report a case which Neches & Hayes (1978) have interpreted as an example of deletion of unnecessary parts. In that case, an advanced subject working thermodynamics problems had difficulties with a very simple problem because he tried to apply his "standard" formula to it without noticing that the equation was inappropriate. (It was only applicable to the more complex problems which the subject normally had to deal with.) Neches & Hayes observed that, "He seemed to have eliminated a test for problem-type which was once part of his strategy for solving physics problems" (pg. 258).

We can see this problem solver's difficulties as an instance in which the predictable-effects pattern was applied. In this case, it was applied inappropriately. I take examples such as this, along with the body of literature on *einstellung* effects exemplified by Luchins' water jug problems (Luchins, 1942; Luchins & Luchins, 1970), as demonstrations of the heuristic nature of the strategy change process. These patterns are useful because they usually lead to good strategy changes, not because they always do.

Some changes are probably safer than others, though. For example, in contrast to eliminating apparently predictable portions of a procedure, this last pattern identifies portions which appear to be redundant.

> A procedure contains two tests as part of a decision, call them T_1 and T_2. It is observed that the tests agree. That is, T_1 is observed to succeed on several occasions, with T_2 also succeeding on all of those occasions, and is observed to fail on several other occasions, with T_2 also failing on all of those occasions.

This pattern suggests that the results of two tests are correlated. If this is the case, then it *may* be that one provides no new information given that the results of the other are available. In that case, we can say that the procedure is *overdetermined*, that is, that one of the tests is unnecessary. This pattern offers a way to further explicate Neches & Hayes (1978) proposed explanation of Neisser's (1964) visual search task. Neisser reported that practiced subjects were twice as fast at locating a particular letter among dissimilar letters as among similar letters. In the dissimilar-letters condition, Neches and Hayes suggested, the procedure for finding the target letter is overdetermined when the letter itself is sought. Many different perceptual tests go into distinguishing one letter from another; if the

surrounding letters are dissimilar, some of those tests become redundant.

2.5.6 Saving partial results

If a result is already available, it is a waste of time to recompute it. When a particular result is frequently needed, it is worthwhile to store it in long-term memory so that it is always available. This is the effect acheived by reduction to results (section 2.5.1).

Sometimes, though, a result is needed many times within a particular problem but the result which is needed varies from problem to problem. In such cases, remembering the result permanently may be neither useful nor feasible. These cases are the domain for which saving partial results is useful.

As a very simple example, consider the task of multyplying two large digits together, say, *43,108 x 32,343*. It turns out that *3 x 43,108* is *129,324*. It is useful to remember this later in the problem when it is necessary to compute *300 x 43,108* and *30,000 x 43,108*. However, it is probably not generally useful to memorize forever the relationship between *3*, *43,108*, and *129,324*. Saving partial results transformations apply to cases such as this, where a result only needs to be saved temporarily. Unlike reduction to results, which utilizes long term memory, saving partial results modifies procedures to make better use of short term memory or external memory.

One pattern which suggests a saving is possible is what might be called the opportunity rule:

> A procedure is about to be executed with a certain input, but the result of that procedure with the same input is recorded in working memory.

It sometimes may be that the information is no longer in working memory. In that case, the pattern is almost, but not quite, the same:

> A procedure is observed with a certain input, but its result is not active in working memory.

These patterns are so straightforward that there is almost nothing to be said about them. The first case does not even really require a modification to the procedure. All that is needed is a general result-copying procedure which is invoked whenever the pattern is noted. The second case suggests some obvious modifications: using external memory or rehearsing the result.

2.5.7 Re-ordering

Changing the order in which operations are performed can reduce both processing and memory demands. It can reduce processing demands because, sometimes, performing an action earlier or later means that other actions do not have to be performed as often. It can reduce memory demands because, often, doing things in a different order can cut the need to remember information for long periods. Neches & Hayes (1978) suggest that the efficient strategies of mental arithmetic prodigies (cf. Hunter, 1968) can be seen as re-orderings of ordinary calculational procedures.

It is relatively easy to think of patterns which suggest exploring re-orderins to reduce memory load. I will present two. The first is quite simple:

> A large number of items are in working memory which have not been used.

Another pattern does not depend on the number of items, but is useful in improving cases involving a single item:

> A result is generated, but many operations intervene before any use of it is made.

These two patterns are correlated, although not completely equivalent. They are correlated because the longer a result has to wait before being used, the more likely it is to contribute to a crowding condition in working memory.

There are a number of patterns which could be stated for re-ordering to reduce processing effort. I will not go into them here, although I'd like to note for those with computer science backgrounds that these patterns are essentially the same as the rules developed for use in optimizing compilers (cf. Allen & Cocke, 1972).

3. A Simple Example: Children's Addition Strategies

I would like to illustrate the theory's application in a simple, familiar domain. In this section present a task analysis of acquiring facility at adding two digits together.

3.1 Defining the initial and final procedures

The first step in an analysis of difficulties in discovering effective strategies is, not surprisingly, to determine what strategies are used. Various reaction time studies suggest that children start with a very simple computational procedure and acquire a more sophisticated procedure, which even adults fall back on when their mature memory retrieval strategies fail.

3.1.1 Precursor of adult procedures: the MIN method

Groen & Parkman (1972) studied solution times on simple addition problems for both children (average age 6 years, 10 months) and adult college students. The data from 19 of the 37 children studied were best explained by assuming that they followed a procedure called the "MIN model". This procedure gets its name from the observation that the time taken to produce an answer depends only on the minimum of the two addends. Essentially, as Groen & Parkman state it, adding x and y by the MIN procedure involves the following steps:

1. Set a counter, "*Value*", to the larger of x and y.
2. Set a counter, "*Increment*", to 0.
3. Increment *Value* by 1.
4. Increment "*Increment*" by 1.
5. If "*Increment*" is less than the smaller of x and y, then go to step 3.
6. Report "*Value*" as the answer.

(Another way of stating this procedure is, "Start with the larger number, and increment it the smaller number of times.")

3.1.2 Successors of the MIN method: memory strategies

Adult performance, Groen & Parkman found, could be explained in two ways. Either: (a) adults always followed the MIN procedure, but were much faster at carrying out some of the steps; or (b) the adults usually retrieved the answer from memory, but 5% of the time suffered from recall failures and were forced to fall back on the MIN procedure. The first

explanation requires that adults improve over children by an implausibly large factor of 20, so we will assume for the remainder of this discussion that the latter explanation is correct.

There is some evidence that memory for addition answers develops at different rates for certain problem types. Groen and Parkman found that slopes for *tie problems* (1+1, 2+2, ...,9+9) in their children's data differed from those predicted by the MIN computational procedure. Although slower than adults, it appeared as if children had some memory for tie problem answers. Woods, Resnick, & Groen (1976) found a similar effect for tie problems when studying single digit subtraction problems.

3.1.3 Precursors of the MIN method: the SUM procedure

If the end product is a fast memory-retrieval process, is the MIN procedure then the initial procedure? It seems not. Groen & Resnick (1977), studying even younger children (average age 4 years, 8 months), provide evidence that the MIN procedure develops from another procedure. This procedure was dubbed the "SUM MODEL", because time to produce an answer grows with the *sum* of the two digits. To add x and y, the SUM procedure follows these steps:

1. Count out x "objects" (e.g., fingers).
2. Count out y objects.
3. Count how many objects there are all together.

As can be seen from the language used to describe the SUM procedure, it is an extremely easy method to communicate. However, since the sum is always greater than the minimum addend, this strategy is far less efficient -- on the average, 5 extra operations will be required (at least 1, and at most 9). These extra operations correspond to the difference between the sum and the minimum operand.

3.2 Characterizing differences between procedures

So far, we have seen that the development of addition procedures can be separated into the development of computational procedures, and the development of memory retrieval procedures. There is some reason to believe that these two developments overlap in time. Since there is little of subtlety to say about the differences between computational and memory retrieval processes, I will focus on differences and transitions between computational methods.

Insert Figure 1 about here

Count Values	Object List
1	A
2	AA
1	AAA
2	AAAA
3	AAAAA
4	AAAAAA
5	AAAAAAA
1	AAAAAAA
2	AAAAAA
3	AAAAA
4	AAAA
5	AAA
6	AA
7	A

A.

Count Values	Increment Values
5	
6	1
7	2

B.

The model of strategy change which drives this task analysis puts a heavy emphasis on short term memory. Let's start the analysis, therefore, by asking what sort of traces the SUM and MIN procedures might leave in working memory. Figure 1 illustrates the two procedures applied to the problem 2 + 5. The SUM procedure is shown in Figure 1a. The left column indicates the values generated for a counting procedure, which is executed three times in the procedure. As can be seen, the counter is incremented first from null to the value of one addend, then from null to the value of the second addend, and finally from null to the value of the answer. Each time a new count is started, the critical information obtained up to that point is represented in an external memory aid, the *Object List* illustrated in the right hand column.

Figure 1b, which illustrates the MIN procedure, is naturally much shorter. One column indicates the sequence of values assigned to the *Value* counter referred to in section 3.1.1, the other indicates the values assigned to the *Increment* counter. As can be seen, the *Value* counter starts with the larger addend, and is incremented to the answer. The *Increment* counter starts with nothing, and is incremented up to the smaller addend.

3.2.1 Comparison of strategies

There are four major differences between the SUM and MIN strategies. SUM has an *Object list*, representing the use of an external memory; MIN does not use external memory. SUM counts out three sets of numbers: (a) from 1 to the first number; (b) from 1 to the second number; and then, (c) from 1 to the sum of the two addends. MIN, on the other hand, counts out two sets of numbers: (a) from 1 to one of the addends; and, simultaneously, (b) the last part of the sum, that is, from the other addend to the total. Thus, the counting operations performed in the MIN procedure can be seen as a *subset of the counting operations performed in the SUM procedure.*

MIN has an *addend selection rule*, which calls for distinguishing the two addends as larger *vs.* smaller. SUM has a random preference rule.

Finally, SUM performs its counting serially, finishing each of its three counts before initiating the next. In the MIN procedure, two counts are interleaved, as the procedure alternates between incrementing the *Value* count and the *Increment*.

What this list of differences amounts to, is an argument that the MIN procedure can be obtained from the SUM procedure by making the following set of changes:

1. Eliminate the Object List.

2. Add a procedure for identifying the larger and smaller addends.

3. Eliminate the addend count-up phase for the larger addend.

4. Eliminate the portion of the total count-up phase in which the larger addend is counted. (So that the count goes from "*max,..., total*" rather than from "*1,...,max, ...total*".)

5. Re-arrange the sequence of operations so that the remaining groups of actions (counting from 1 to the smaller addend, and counting from the larger addend to the sum) are interleaved.

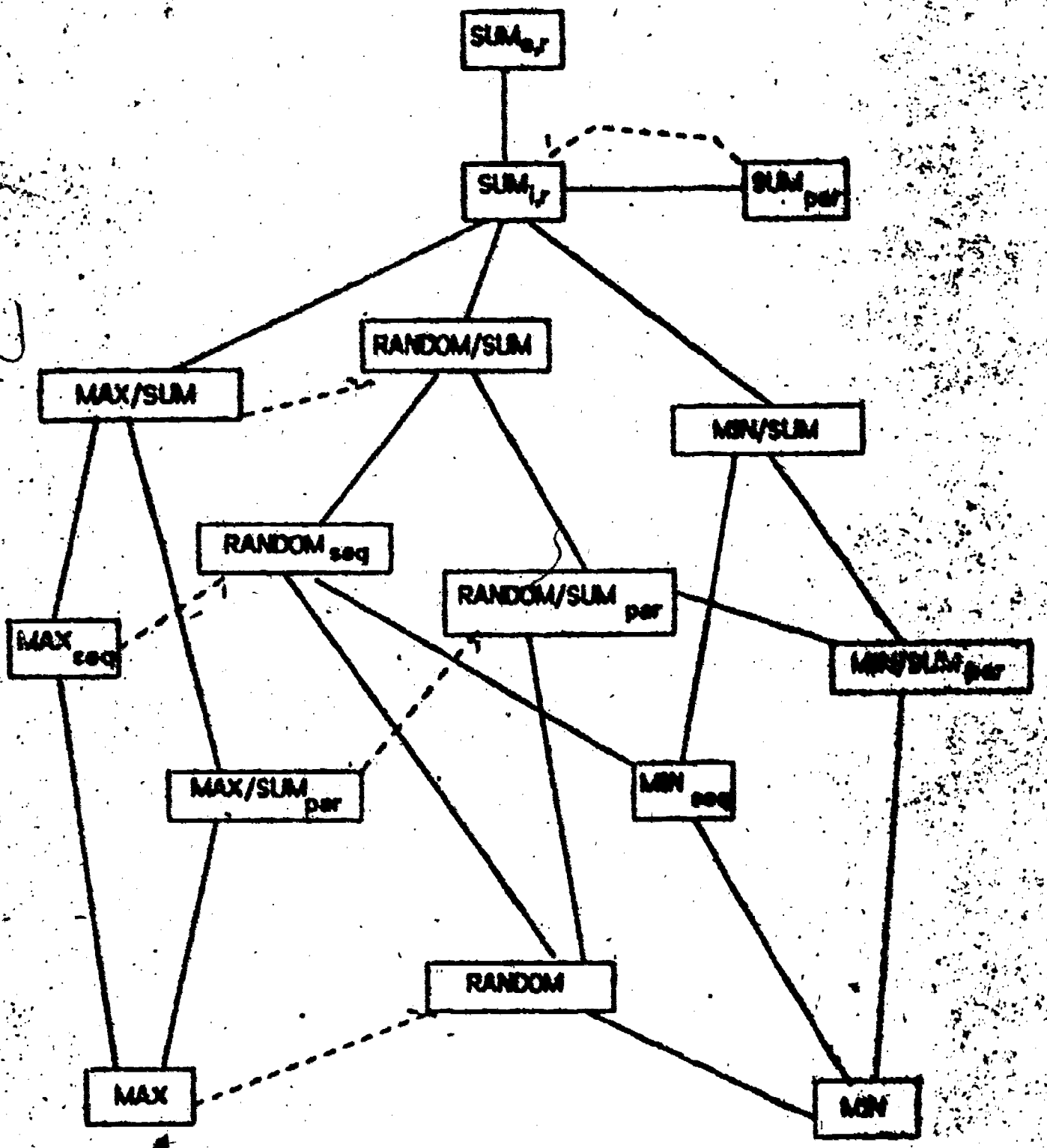
To see exactly why these changes are sufficient, it is necessary to have a tighter information processing analysis of the two procedures than has been provided so far. It should also be noted that these changes are only *partially ordered*. That is, there is no *a priori* reason to expect them to be made simultaneously, or even in any unique order. Section 3.2.2 is concerned with these two points. It establishes the background for the analysis of specific transitions in section 3.3.

3.2.2 The space of strategy variants

Although it may seem that the SUM and MIN strategies have already been described in detail, the fact is that a number of issues have been left unspecified. For example, nothing has been said about commutivity, or the concept of greater/lesser, even though both are implied in the description given for the MIN procedure. Nor have the nature of tests and operations been specified, e.g., how the procedure determines when to stop counting. (This, I will be claiming, requires the concept of one-to-one correspondance.)

When these details are specified, and the independence of possible strategy changes is taken into account, we will see that what appears is not a single SUM method and a single MIN method. Rather, what will be found is a *space* of addition methods, consisting of *families* of related strategies. Some of these strategies belong to the SUM family, some to the MIN family, and others fall somewhere in between. Figure 2 represents this space as a network structure, in which nodes represent different procedures and arrows represent strategy transformations leading from one procedure to another. This section will try to formally specify these procedures, and informally justify the transitions between them. Section 3.3 will provide a more formal analysis for some selected transitions.

Insert Figure 2 about here


SUM$_{seq}$
SUM$_{par}$
RANDOM/SUM
MAX/SUM
MIN/SUM
RANDOM$_{seq}$
RANDOM/SUM$_{par}$
MIN/SUM$_{par}$
MAX$_{seq}$
MAX/SUM$_{par}$
MIN$_{seq}$
RANDOM
MAX
MIN

Let us start with the SUM strategy. An informal description was given in section 3.1.3. Figure 1a has illustrated the sequence of values generated in course of following some version of the procedure. What has not been specified are: (a) the rules for selecting which addend to start working with first; (b) the rule specifying when to stop counting; (c) the mental operations which implement these rules; and (d) the mental representation of information which is operated upon. I will start by describing $SUM_{e,r}$, a basic version of the SUM family. A program for it is defined in Figure 3.

Insert Figure 3 about here

$SUM_{e,r}$ has a first-noticed rule for selecting addends, which says that the addend of first interest is determined randomly. Its rules controlling counting of addends represent a form of one-to-one correspondance; for each number mentioned, an object is added to the external memory. Its rules controlling counting of the total represent the converse one-to-one correspondance; each time an object is found, the next number is generated. In both cases, the processes start with an initial value, and then run until interrupted. When counting addends, the interruption stops the counting when the count matches the addend. When counting the total, the interruption occurs when the procedure runs out of things to count. The mode of representation is partly auditory, since numbers are spoken or sub-vocalized, and partly visual, since the external display of objects must be scanned.

Among the key features of $SUM_{e,r}$ is its reliance on an external memory. The counting out of both addends leaves one number in working memory for each object in the external memory. To count up the total, it's sufficient to count the *number of numbers* counted out for the addends. Therefore, external memory can be eliminated in favor of reliance on the information in working memory. The resulting procedure, $SUM_{i,r}$ (for internal, random), is shown in Figure 4. As can be seen, it is identical to the $SUM_{e,r}$ procedure shown in Figure 3, with the exception of changes to steps 4 and 12.

Insert Figure 4 about here

The next series of transitions delete redundant counting operations in two stages. One stage eliminates the counting of one of the addends. This produces procedures which count out the entire total, but only one of the addends. Their performance characteristics would depend on both the sum and the counted addend. The addend selection rule determines which will be the critical addend, thus providing variants RANDOM/SUM, MIN/SUM, MAX/SUM. Figure 5a presents a program for RANDOM/SUM; it is essentially identical to Figure 4, with

FIGURE 3

SUM
e,r

1) Pick an addend at random; call it X.

2) Call the other addend Y.

3) Put a 1 in working memory.

4) Place an object in the sequence held in external memory.

5) Compare the number in working memory to X.

6) If the two are not equal, put the number's successor in working memory, and go to step 4.

7) Put a 1 in working memory.

8) Place an object in the sequence held in external memory.

9) Compare the number in working memory to Y.

10) If the two are not equal, put the number's successor in working memory, and go to step 8.

11) Put a zero in working memory.

12) Try to fetch an object from external memory.

13) If an object is found, put the number's successor in working memory, and go to step 12.

14) Report the last number in working memory as the answer.

FIGURE 4

SUM
i,r

1) Pick an addend at random; call it X.

2) Call the other addend Y.

3) Put a 1 in working memory.

5) Compare the number in working memory to X.

6) If the two are not equal, put the number's successor in working memory, and go to step 5.

7) Put a 1 in working memory.

9) Compare the number in working memory to Y.

10) If the two are not equal, put the number's successor in working memory, and go to step 9.

11) Put a zero in working memory.

12) Try to fetch a count element from working memory

13) If an object is found, put the current number's successor in working memory, and go to step 12.

14) Report the last number in working memory as the answer.

the exception of the steps relating to the addend count.

Insert Figure 5 about here

The second stage eliminates the first portion of the total count, which corresponds to the same addend eliminated in the first stage. This change creates procedures which count out one addend and then count from the other addend to the total, maintaining a 1-1 correspondance with the count for the first addend. These procedures will be sensitive only to the first addend. Depending on the selection rule, we will get $RANDOM_{seq}$, MIN_{seq}, or MAX_{seq}. Figure 5b specifies $RANDOM_{seq}$.

However, another change becomes feasible at about the same time as the second stage of deleting unnecessary parts. This other change is to re-order the sequence of operations so that the addend count is done *during* the total count, rather than before it. This offers the advantage of reduced memory load, since digits in the addend count are used immediately instead of being held for later use.

The second deletion and the re-ordering, are independent. Theoretically, they could take place in either order, so the combinatorics generate a set of related variants. If the re-ordering occurs after the deletion, then it transforms the sequential strategy into its corresponding parallel counting procedure: RANDOM, MAX, or MIN. If the re-ordering is discovered first, we get variants of the /SUM procedures: $RANDOM/SUM_{par}$, MIN/SUM_{par}, and MAX/SUM_{par}. Making the deletion then leads to procedures RANDOM, MAX, or the goal procedure: MIN. Thus, either path leads to RANDOM, MAX, or MIN.

Figure 6 illustrates the difference between sequential and parallel procedures by specifying MIN_{seq} and its counterpart, the parallel procedure MIN (which also happens to be the Holy Grail of this expedition). Comparing the MIN_{seq} procedure of Figure 6a with the $RANDOM_{seq}$ procedure shown in Figure 5b will show that they differ only in step 1, the addend selection rule. In turn, the MIN procedure of Figure 6b differs only in the arrangement of steps.

Insert Figure 6 about here

If the path has lead to MIN, of course, then the procedure's development is essentially complete. If not, then the transitions of interest are those which lead to members of the MIN family. Note that, in Figure 2, these transitions are all represented as going through the RANDOM family of procedures. In this analysis, the MAX family branch of the tree is viewed

A. RANDOM/SUM

1) Pick an addend at random; call it X.

2) Call the other addend Y.

3) Put a 1 in working memory.

5) Compare the number in working memory to X.

6) If the two are not equal, put the number's successor in working memory, and go to step 5.

7) Put a 1 in working memory.

9) Compare the number in working memory to Y.

10) If the two are not equal, put the number's successor in working memory, and go to step 9.

11') Find the last number of the first count sequence; make that the current number in working memory.

12) Try to fetch a count element from working memory.

13) If an object is found, put the current number's successor in working memory, and go to step 12.

14) Report the last number in working memory as the answer.

B. RANDOM
seq

1) Pick an addend at random; call it X.

2) Call the other addend Y.

7) Put a 1 in working memory.

9) Compare the number in working memory to Y.

10) If the two are not equal, put the number's successor in working memory, and go to step 9.

11'') Make X the current number in working memory.

12) Try to fetch a count element from working memory.

13) If an object is found, put the current number's successor in working memory, and go to step 12.

14) Report the last number in working memory as the answer.

A. MIN
seq

1') Pick the LARGER addend; call it X.

2) Call the other addend Y.

7) Put a 1 in working memory.

9) Compare the number in working memory to Y.

10) If the two are not equal, put the number's successor in working memory, and go to step 9.

11'') Make X the current number in working memory.

12) Try to fetch a count element from working memory.

13) If an object is found, put the current number's successor in working memory, and go to step 12.

14) Report the last number in working memory as the answer.

B. MIN

1') Pick the LARGER addend; call it X.

2) Call the other addend Y.

11'') Make X the "key" number in working memory.

7) Put a 1 in working memory. (Call it the "other".)

9) Compare the other number in working memory to Y.

13') Put the key number's successor in working memory.

10) If the other number doesn't equal Y, put its successor in working memory, and go to step 9.

14) Report the last number in working memory as the answer.

as a blind alley (if the reader will forgive a mixed metaphor). In the next section, we will see why this is so.

3.2.3 Summary of results

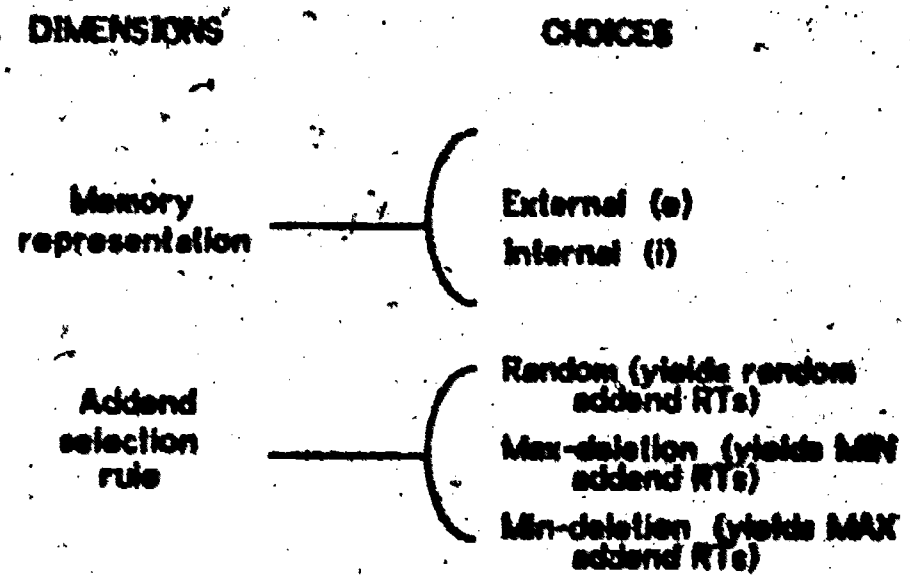
At this point, we have compared a simple initial SUM strategy to a sophisticated "expert" MIN strategy. The results of this comparison are summarized in Figure 7. All addition procedures rely on procedures for counting and for maintaining one-to-one correspondance between two sets of symbols. However, procedures essentially differ along four dimensions:

1. *Memory representation*: whether or not external memory aids are needed.
2. *Addend selection rule*: the criteria determining which addend will receive special handling.
3. *Sets of objects counted*: the number of times a counting process is invoked, and the starting and ending points of each count.
4. *Ordering of counting operations*: whether the counting operations are performed separately or concurrently.

Insert Figure 7 about here

The initial SUM strategy uses external memory, counts three sets of objects, and performs the counts sequentially (cf., Figures 1a and 3). The goal MIN strategy uses only internal memory, counts two sets of objects, performs the counts concurrently, and has a larger-addend selection rule (cf., Figures 1b and 6b).

An analysis of the patterns inherent in traces of the SUM procedure has lead to a set of intermediate alternatives along each of the four dimensions. Since the four dimensions are almost independent, this in turn led to the hypothetical space of related addition procedures shown in Figure 2. For convenience of exposition, I have presented the analysis in reverse order, by presenting the space of procedures before discussing the patterns which caused me to generate it. This allows me to present the strategy transformations in terms of groups of similar transitions, rather than having to consider each transition separately. These groups are the topic of section 3.3.


DIMENSIONS
CHOICES
Memory representation
External (e)
Internal (i)
Addend selection rule
Random (yields random addend RTs)
Max-deletion (yields MIN addend RTs)
Min-deletion (yields MAX addend RTs)
Items counted
2 addends, total
1 addend, total
1 addend, portion of total
Ordering of counting operations
Sequential (seq)
Parallel (par)

3.3 Analyzing SUM-to-MIN transitions

This section will discuss three sets of transitions: (a) from SUM to the three sequential procedures; (b) from sequential procedures to MIN_{seq}; and, (c) from sequential procedures to parallel procedures.

3.3.1 The elimination of addend counts

Although procedures in the SUM family differ in their addend selection rules, there are only three cases which need to be considered for the entire family. Either the first addend selected is the larger of the two (call it *Max*), the smaller of the two (call it *Min*), or the addends are equal. Figure 8a illustrates the case where *Min* precedes *Max*, Figure 8b the case where *Max* precedes *Min*.

Insert Figure 8 about here

In the *Min*-first case: (a) the addend count goes from 1 to *Min*; (b) starts at 1 and passes through *Min* on the way to *Max*; then, (c) starts at 1 and passes through both *Min* and *Max* on the way to the sum. In the *Max*-first case, (a) and (b) are reversed. Since the total count is kept in correspondance with the digits generated by the two addend counts, each digit in the total count is paired with a digit in the addend count.

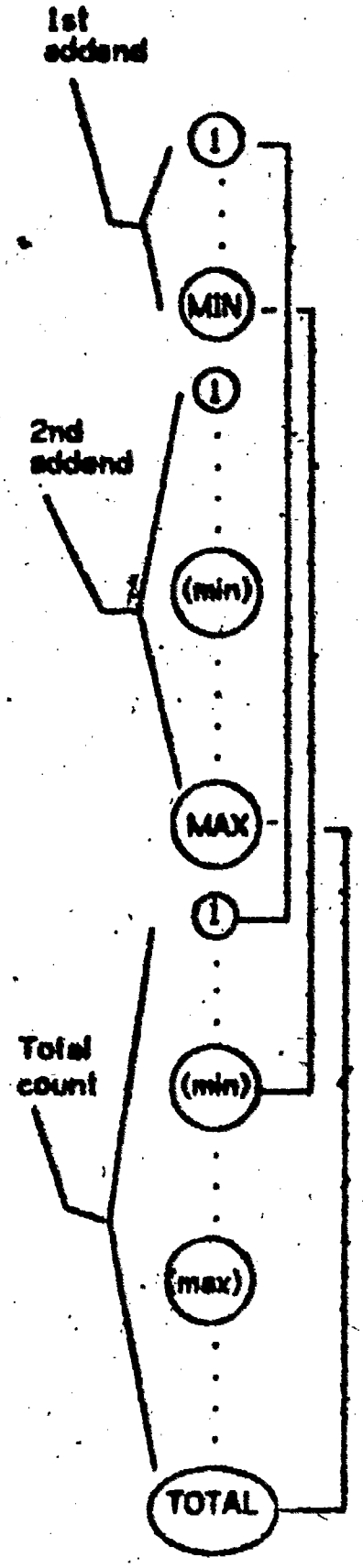
The most striking pattern is that, in all cases:

> The first portion of the total count is exactly identical to the items being counted, the digits of the first addend.

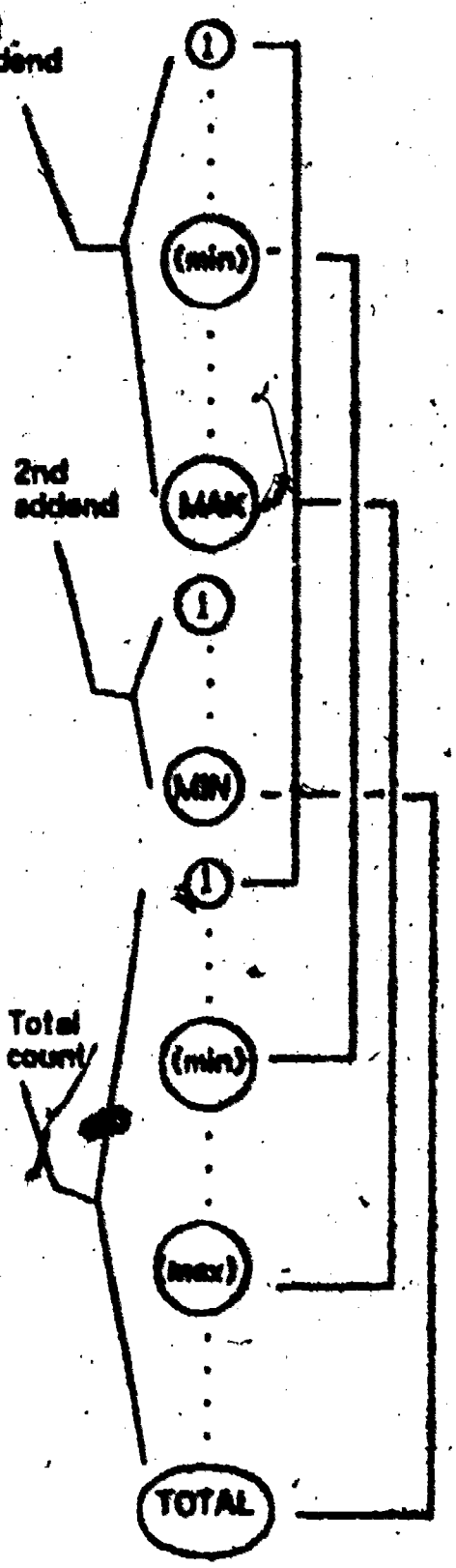
This sort of pattern, where the input of an operation is the same as its result, was one of the patterns which section 2.5.5 suggested as a trigger for deletion of unnecessary parts.

There are actually three possible outcomes of this pattern. The most likely would be to describe the identity as being between the *selected addend* and the total count. This description accounts for all instances of the pattern, and suggests deleting the unnecessary portion of the total count. Since this change leaves the current addend selection rule intact, the transition would convert $SUM_{i,r}$ into RANDOM/SUM. Just below, I'll discuss the paths from there to a sequential strategy, but first let's consider the two less likely descriptions of the pattern. These would lead directly to MIN/SUM or MAX/SUM.

A.


1st addend
1
MIN
1
2nd addend
(min)
MAX
1
Total count
(min)
(max)
TOTAL

B.


1st addend
1
(min)
2nd addend
MAX
1
MIN
1
Total count
(min)
(max)
TOTAL

Assume that a child executing a SUM procedure had described the addends in terms of relative size (i.e., as *Min* and *Max*). Then, the patterns available for observation divide into three sets: *Min*-first, *Max*-first, and tie problems. Under a random addend selection rule, and almost any distribution of problems, the probability is 45% that the pattern instance which is first noticed will be *Min*-first. (The probability is the same for *Max*-first. Tie problems constitute the remaining 10%.) Thus, there is some small probability of describing the pattern as indicating that it is unnecessary to perform the portion of the total count corresponding to *Min* (or *Max*). This is quite different from focussing on the first digit. These descriptions suggest modifying the addend selection rule, in addition to deleting some counting operations. Depending on whether *Min* or *Max* was deleted, the result would be to transform $SUM_{i,r}$ into MAX/SUM or MIN/SUM, respectively. (The reversal comes because the resulting procedure is sensitive to the remaining addend, rather than the deleted addend.)

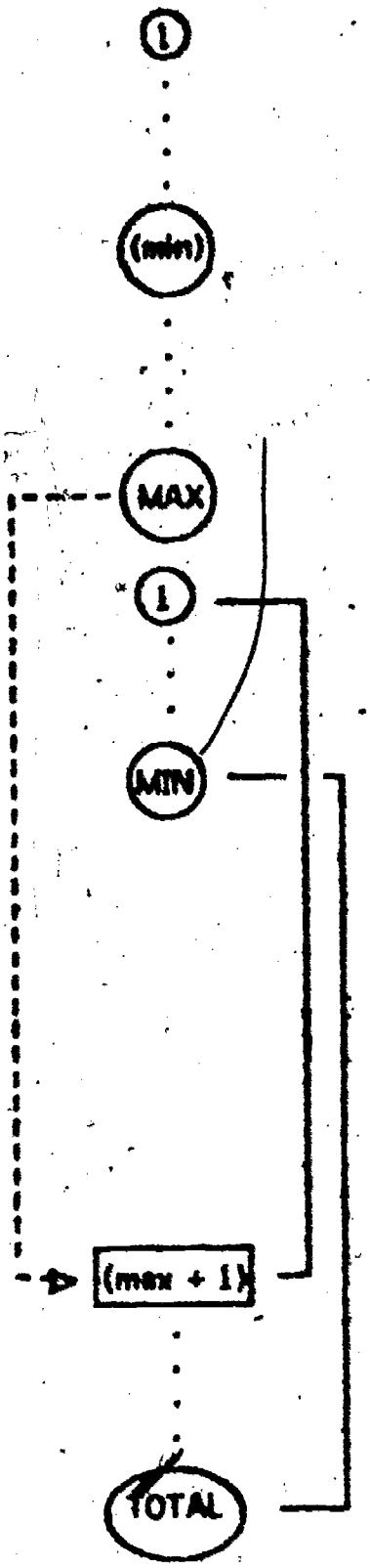
There are also several other patterns which, although even less likely, would support a direct transition to MIN/SUM or MAX/SUM. These involve comparing the second addend count with the total count. Although the second addend count occurs after the first, it is further away in memory from the total count. This is because the digits of the first addend count are being brought forward for pairing with digits in the total count. These patterns, although more distant, are of the same form as the preceding ones:

> For *Min*-first problems, the total count will be identical with the second addend all the way through *Max*.
>
> Similarly,
>
> For *Max*-first problems, they will be identical up to *Min*.

Thus, the two patterns are the mirror images of the primary patterns. The distance between elements in them is the factor which makes them secondary. Rather than involving an identity between contiguous elements, these patterns involve an identity between items separated by a distance proportional to the size of the addend. Once again, these patterns suggest a deletion of part of the total count. If the addends are represented as *Min* and *Max*, in addition to first and second, then the pattern also suggests a switch in addend selection rule which leads to MIN/SUM or MAX/SUM.

Insert Figure 9 about here

A.



B.

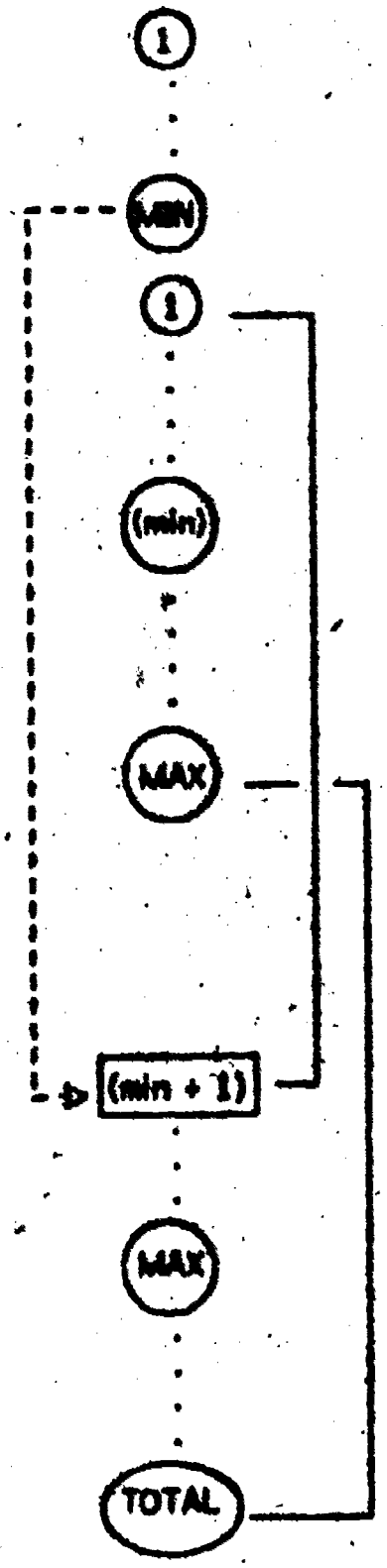


Figure 9a shows the sequence of operations involved in the MIN/SUM procedure; Figure 9b shows MAX/SUM. RANDOM/SUM will alternate between behaving like MIN/SUM or MAX/SUM. There is one general pattern present in all /SUM procedures which suggests a transition to the corresponding sequential strategy:

> The portion of the total count corresponding to the second addend count is used only to generate the addend itself, although that addend is already known (since it is given in the problem).

This is a pattern associated with saving of partial results (section 2.5.6), which suggests replacing that portion of the total count with the addend given in the problem (which is the end result of that portion). This change, since it does not consider addend selection rules, moves whichever /SUM strategy is in use over to the corresponding sequential strategy.

As will be seen below, the only competition faced by this pattern comes from patterns which suggest other useful changes. The procedures produced by making those other changes generate patterns equivalent to this one. Thus, the difficulty of making this transition seems relatively small. The difficulty might be even smaller if a transition to a parallel procedure (cf. section 3.3.3) has already been made.

3.3.2 Developing the correct addend selection rule

In order to have the most efficient procedure, MIN, it is necessary to have an addend selection rule which focuses on the larger addend. When the addend count for the *Max* digit is deleted, and the total count is started from *Max*, then all steps which can be eliminated will have been.

This section will consider transitions which lead from members of the RANDOM and MAX families into members of the MIN family. (Remember that these first two families have addend selection rules which attend to the first-noticed addend, or the smaller addend, respectively.) As Figure 2 illustrates, there are many points at which the transition to a larger-addend selection rule can be made. This section will consider first the transitions from a first-noticed rule to a larger-addend rule (i.e., RANDOM family to MIN family), then transitions from a smaller-addend rule to a larger addend rule (i.e., MAX family to MIN family).

The patterns which suggest modifying the first-noticed addend selection rule exist because procedures in the RANDOM family sometimes behave like the corresponding MAX family member and sometimes behave like the corresponding MIN family member. This, of

course, is because the first-noticed addend has an equal likelihood of being the larger or smaller of the two. As a result, it is possible to compare sequences generated when solving equivalent problems at different times.

If we consider the sequences of actions generated by RANDOM/SUM, we see that:

> The two sequences of operations shown in Figure 2 produce the same answers, but one involves less effort. (The effort difference is proportional to the difference between the addends.)

The sequences generated by RANDOM/SUM$_{par}$ (see section 3.3.3), although differing in the order in which operations are performed, also have the same properties as described in this pattern. In RANDOM$_{seq}$ and the pure RANDOM procedure, the same type of pattern also appears. However, because an additional set of counting operations has been deleted, the effort difference is more significant: it is now proportional to *twice* the difference between *Max* and *Min*.

The pattern is one associated with reduction to a rule (section 2.5.2). It suggests looking for a correlate of the effort difference, and using that correlate to pick the method for solving the problem. Thus, the pattern is useless unless the addends are described in terms of greater and lesser; otherwise nothing can be found which correlates with the effort difference. If this description is available, however, the change suggested by the pattern is a new addend selection rule which marks the larger addend for deletion. This rule, of course, converts a RANDOM family member to the corresponding MIN family member.

Note that, although this is a highly critical transition in reaching the goal MIN procedure, it is also a highly difficult transition. This pattern compares action sequences across different problems, rather than within problems. Noticing the pattern requires that equivalent problems appear reasonably close to each other. If problems are selected at random, the probability of this happening is relatively low. For any given problem, the chance of the equivalent problem occuring within the next five problems is less than 0.05.

On first inspection, it would seem that this analysis suggests that it would be useful to give students pairs of equivalent problems. There are, however, some difficulties in such a simple approach. If a sequence of problems is given where equivalent problems are deliberately placed close together, then competition is created from another strategy change: simply copying the result of the first problem, without doing any computation whatsoever on the second problem. This competing strategy is much less general, since it is only of use when a problem is equivalent to a recently preceding one. Nevertheless, it would preclude

the comparison of effort differences necessary to switch to a MIN strategy.

The transition from RANDOM family membership to MIN family membership is particularly important because the total task analysis suggests that most of the paths to MIN go through some RANDOM procedure. MAX family members lack patterns directly suggesting transitions to MIN. By looking at the portions representing MAX-family procedures in Figures 8b and 9b, we can see a set of patterns appearing in the assorted MAX variants:

> If the *Max* addend is still counted out, then
> its count will match with the total count.

This pattern is of the same form as patterns mentioned earlier. However, it only applies to action sequences generated by deleting the count of the *Min* addend (e.g., in MAX/SUM). The exact form of one of these sequences depends on which procedure variant is being considered. (Actually, there is a set of related patterns. Since their implications are the same, though, it's convenient to generalize and talk as if there was a single pattern.)

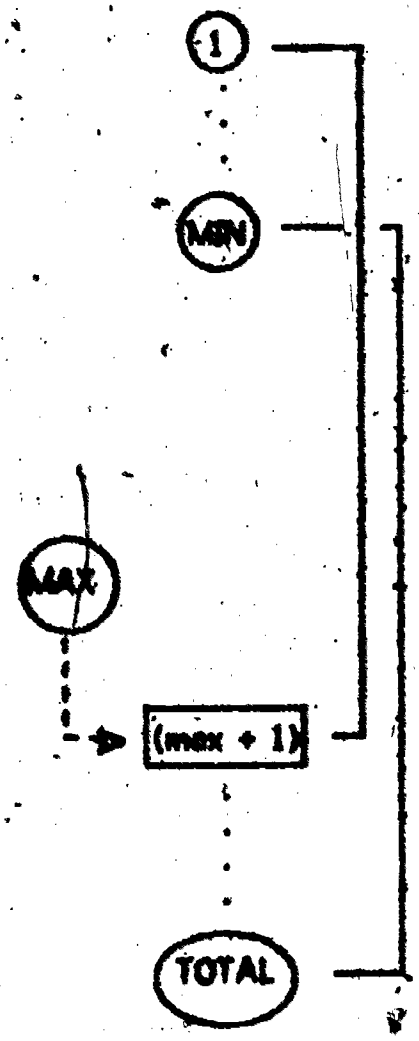
Unlike other patterns of its form, it cannot directly suggest a change. This is because deletion of the *Min* addend count has removed information necessary to constructing a strategy where the *Max* addend count is deleted. Thus, this pattern can suggest that MAX/SUM is unsatisfactory, but not exactly how to modify it. A procedure with sufficient information to enable construction of the MIN/SUM strategy can only be reached by *returning* from MAX/SUM back to RANDOM/SUM. In addition, it should be noted that this is a difficult pattern to detect, since (as Figure 9b illustrates) the portion of the total count between *Min* and *Max* will not be paired with the critical numbers in the addend count.

3.3.3 The switch to parallel counting

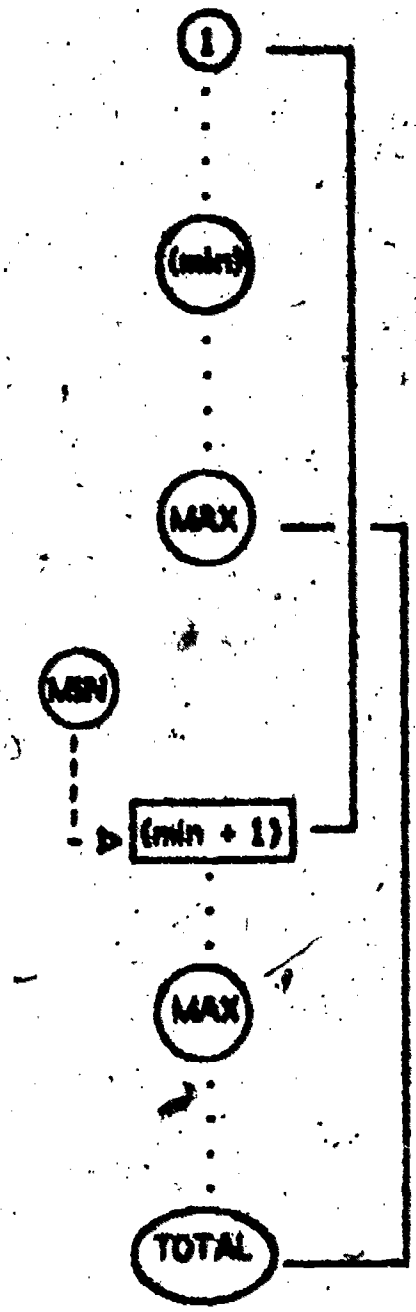
To be maximally efficient, the addition procedure needs to count digits concurrently, rather than counting out each set completely before going on to the next. Concurrent counting drastically reduces both the number of items which must be kept in working memory, and the length of time for which they are needed. This is because information is used as soon as it is produced in the parallel procedures, but has to wait for some time before being used in the sequential procedures. To see this, consider the difference between MIN and its sequential variant MIN_{seq} (cf. Figure 10). If we consider the points of peak memory load in the two procedures, large differences appear.

Insert Figure 10 about here

A.



B.



In MIN_{seq}, the peak comes when the addend has been counted out, but the corresponding total count has not started. At that point, *2 + Min* items are required: (a) the *Max* addend, which is needed to initialize the total count; (b) the *Min* addend, which is needed to test for completion of the addend count; and, finally, (c) items representing each digit in the addend count, which are needed for managing the total count.

With concurrent strategies, such as MIN, there is no such peak. At all points in time, only a small, constant, number of items are required: (a) the addend value used to stop counting; (b) the current value of the total count; and (c) the current value of the addend count.

Section 2.5.7 suggested two patterns associated with re-ordering transformations. It is useful to look for cases where many operations intervene between producing some information item and using it. It is also useful to look at cases where the demand on working memory is high. Both of these patterns apply to the sequential strategies. In justifying the value of making this change to MIN_{seq}, we saw that generating all addend digits before counting any of them cluttered working memory, and led to a delay between generation and utilization proportional to the size of the addend.

It is interesting to note that the need for a re-ordering lessens as the procedure becomes more sophisticated. In, say, the SUM procedures, the generation/utilization distance and the peak memory load are both proportional to the *sum* of the addends. In MIN_{seq}, both factors were proportional not to the sum, but only to the *Min* addend.

4. Closing notes

This paper has tried to take apart the task of simple addition and then put it back together again. It is always somewhat frightening to discover how complicated "simple" tasks really are. In this analysis, things have become quite complicated indeed. What has been gained from the exercise?

First of all, an understanding of the complexity of such a task is useful in itself. It really should not be that surprising that there are complex features in the addition task. This is a skill which takes quite some time for children to master. Since we all believe that our children are terribly bright, we should hope that there is a good reason why it takes so long.

This task analysis has suggested several reasons. I have started out by identifying an efficient strategy and a simple, easily teachable, novice strategy. When we consider the differences between these procedures, we see that a number of independent discoveries must be made. When we consider what is involved in making those discoveries, we see that moving directly to the expert strategy is possible but unlikely. The major alternatives each present difficulties. One set of strategies represent a dead end: the improvements which are possible in that set turn out to eliminate information which is crucial to discovering the best strategy. Fortunately, there is information still available which indicates that a better strategy is possible.

The other major alternative set of strategies contains information which is essential to one of the key discoveries leading to an expert strategy. This information has to do with differences in effort on equivalent problems. The analysis showed that this information might be difficult to acquire naturally, since the appropriate circumstances can be expected to occur only rarely under random conditions. The analysis also warned of difficulties in trying to arrange an appropriate sequence of practice problems. Trying to make the properties of equivalent problems salient by assigning them as pairs, for example, creates a pattern which suggests another shortcut. That shortcut, it turns out, can preclude noticing of the critical properties of the problems.

The emphasis of the analysis has been on patterns. This has been useful in analyzing addition, because it has led to suggestions about a set of related (but still very different) procedures for doing the task. This set has appeared as a result of the simple observation that any complex sequence of actions contains a number of different patterns. Each pattern can suggest a different way to modify that sequence.

One of the great bugaboos of psychology is the current emphasis on aggregated data and group models. In instruction, a critical concern is (or should be) with the sources of variation in individual performance. The approach to task analysis I have tried to present

here is oriented to addressing that concern. Rather than suggesting that there is a single learning path, which all students follow at varying rates, this analysis suggests that we can identify many different paths. Some paths may be better than others, but each presents its own unique difficulties. To optimally gear instruction to a student, it is important to be able to assess his or her individual problems.

Analyses such as I have presented, which help to identify the set of possible strategies and the difficulties entailed in developing expertise with them, are a step in this direction.

References

Allen, F.E., & Cocke, J. A catalogue of optimizing transformations. In R. Rustin (Ed.), Design and optimization of compilers. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1972.

Anderson, J.R. Language, memory, and thought. Hillsdale, NJ: Lawrence Erlbaum Associates, 1976.

Anzai, Y. Learning strategies by computer. In Proceedings of the Second National Meeting of the Canadian Society for Computational Studies of Intelligence, Toronto, 1978.

Baylor, G.W., & Gascon, J. An information processing theory of aspects of the development of weight seriation in children. Cognitive Psychology, 1974, 6, 1-40.

Bhaskar, R. & Simon, H.A. Problem solving in semantically rich domains: an example from engineering thermodynamics. Cognitive Science, 1977, 1(2), .

Collins, A.M., & Loftus, E.F. A spreading activation theory of semantic processing. Psychological Review, 1975, 82, 407-428.

Davis, R., & King, J. An overview of production systems. Computer Science Department, Stanford University, 1975.

Ernst, G.W., & Newell, A. GPS: a case study in generality and problem solving. New York: Academic press, 1969.

Feigenbaum, E.A. The simulation of verbal learning behavior. In E.A. Feigenbaum & J. Feldman (Eds.), Computers and thought. New York: McGraw-Hill, 1963.

Forgy, C. The efficiency of production system implementations. Computer Science Department, Carnegie-Mellon University. Unpublished doctoral disseration, 1978.

Gerrilsen, R., Gregg, L.W., & Simon, H.A. Task structure and subject strategies as determinants of latencies (CIP #292). Pittsburgh: Department of Psychology, Carnegie-Mellon University, 1975.

Groen, G.J., & Parkman, J.M. A chronometric analysis of simple addition. Psychological Review, 1972, 79, 329-343.

Groen, G.J., & Resnick, L.B. Can preschool children invent addition algorithms? Journal of Educational Psychology, 1977, 69, 645-652.

Guthrie, E.R. The psychology of learning (rev. ed.). New York: Harper, 1952.

Hull, C.L. Principles of behavior. New York: Appleton-Century-Crofts, 1943.

Hunter, I.M.L. Mental calculation. In P.C. Wason & P.N. Johnson-Laird (Eds.), Thinking and reasoning. Baltimore: Penguin Books, 1968.

Klahr, D. A production system for counting, subitizing, and adding. In W.G. Chase (Ed.), Visual information processing. New York: Academic Press, 1973.

Klahr, D., & Wallace, J.G. The development of serial completion strategies: an information processing analysis. British Journal of Psychology, 1970, 61, 243-257.

Klahr, D., & Wallace, J.G. Cognitive development: an information processing view. Hillsdale, NJ: Lawrence Erlbaum Associates, 1976.

Koffka, K. Principles of gestalt psychology. New York: Harcourt BRace, 1935.

Lenat, D.B., & McDermott, J. Less than general production system architectures. In Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Boston, 1975.

Levin, J.A. PROTEUS: an activation framework for cognitive process models. Psychology Department, University of California at San Diego, unpublished doctoral dissertation, 1976.

Lewis, C.H. Production system models of practice effects. Department of Psychology, University of Michigan at Ann Arbor, unpublished doctoral dissertation, 1978.

Luchins, A.S. Mechanization in problem solving: the effect of *einstellung*. Psychological Monographs, 1942, Whole No. 248.

Luchins, A.S., & Luchins, E.H. Wertheimer's seminars revisted: problem solving and thinking. Albany: State University of New York, Faculty-Student Association, 1970.

Neches, R. Strategy modification. Pittsburgh, pA: Department of Psychology, Carnegie-Mellon University, manuscript submitted for publication, 1979.

Neches, R., & Hayes, J.R. Progress towards a taxonomy of strategy transformations. In A.M. Lesgold, J.W. Pellegrino, S. Fokkema, & R. Glaser (Eds.), Cognitive psychology and instruction. New York: Plenum Books, 1978.

Neisser, U. Visual search. Scientific American, 1964, 210(6), 94-102.

Neves, D.M. A computer program that learns algebraic procedures by examining examples and by working test problems in a textbook. In Proceedings of the Second Annual Conference of the Canadian Society for Computational Studies of Intelligence, Toronto, 1978.

Newell, A. HARPY, production systems, and human cognition. In R. Cole (Ed.), Perception and production of fluent speech. Hillsdale, NJ: Lawrence Erlbaum Associates, in press, 1979.

Newell, A. Production systems: models of control structures. In W.G. Chase (Ed.), Visual information processing. New York: Academic Press, 1973.

Newell, A., & Simon, H.A. Human problem solving. Englewood Cliffs, NJ: Prentice-Hall, 1972.

Ohlsson, S. Production system reconstructions of theories for the solving of three-term series tasks.

Umea, Sweden: Department of Psychology, University of Umea, 1977.

Resnick, L.B. Task analysis in instruction. In D. Klahr (Ed.), Cognition and Instruction. Hillsdale, NJ: Lawrence Erlbaum Associates, 1976.

Simon, H.A. The functional equivalence of problem solving skills. Cognitive Psychology, 1975, 7, 268-288.

Sternberg, S. The discovery of processing stages: extension of Donder's method. In W.G. Koster (Ed.), Attention and Performance II. Amsterdam: North-Holland Publishing Company, 1969.

Thorndike, E.L. The psychology of learning. New York: Teachers College, 1913.

Waterman, D.A. Adaptive production systems. In Proceedings of the Fourth International Joint Conference on Artificial Intelligence, 1975.

Waterman, D.A. Serial pattern acquisition: a production system approach. In C.H. Chen (Ed.), Pattern recognition and artificial intelligence. New York: Academic Press, 1976.

Waterman, D.A., & Hayes-Roth, F. Pattern-directed inference systems. New York: Academic Press, 1978.

Waugh, N.C., & Norman, D.A. Primary memory. Psychological Review, 1965, 72, 89-104.

Woods, S.S., Resnick, L.B., & Groen, G.J. An experimental test of five process models for subtraction. Journal of Educational Psychology, 1975, 67, 17-21.

Young, R.M. Children's seriation behavior: a production system analysis. Pittsburgh: Department of Psychology, Carnegie-Mellon University, unpublished doctoral dissertation, 1973.